Security & Risk Report

1. Hill Cipher:

a. Prerequisite Math Background:

So before going into explaining the Hill cipher algorithm, let's just go through some math concepts that will be useful when explaining the Hill cipher algorithm

- Matrices:
  - a matrix can be thought of as a 2d array in programming languages, simply a table containing multiple rows where each row has the same number of elements. So we can define a matrix dimensions as rows * columns
  - Matrix multiplication involves multiplying the elements of rows of the first matrix by the elements of columns of the second matrix and summing the results this implies that the column size of the first matrix has to be equal to the row size of the second matrix
    - formally defined as:
    - given matrix **A** with dimensions **NxM**
    - matrix **B** with dimensions **MxK**
    - the resulting matrix **C = AB** where $C_{ij} = \sum_{l=1}^{m} (a_{il}. b_{lj})$, and **C** have dimensions **NxK**
  - A square matrix is a matrix where the number of rows is equal to the number of columns
  - One of the most popular square matrices is called the Identity matrix, usually denoted with **I**, the identity is a matrix where all elements in the matrix are equal to zero except the elements at the diagonal are equal to 1
    - example 3x3 identity matrix
      [1 0 0]
      [0 1 0]
      [0 0 1]
  - When the identity matrix is multiplied by any other matrix A, it produces the same matrix A
    - **IA = AI = A**
    - where dimensions of **A** is **NxN** and dimensions of **I** is **NxN**
  - For a square matrix A it will have an inverse A^{-1} if and only if the matrix multiplication of both matrices $A.A^{-1}$ yields the identity matrix I
    - $A.A^{-1} = A^{-1}.A = I$
  - A vector is a special type of matrix such that it has only a single row or only a single column
    - example a vector **V** can have dimensions $1 \times N$, or dimensions $N \times 1$
- Modular Multiplicative Inverse:
  - For a given integer a and a modulus m, the modular multiplicative inverse $(a^{-1}.a)\ \%\ m = 1$
  - this can also be written as $(a^{-1}.a - 1)\ \%\ m = 0$
  - by the way, it is not guaranteed to find a^{-1} given an integer a and a modulus m

b. Task Goal & Algorithm:

The task goal was to encrypt the 26 upper-case ASCII letters in plain text using the Hill cipher encryption algorithm. Let's define how the Hill cipher algorithm works:

- Key Generation
  - the first step of the Hill cipher algorithm is generating a random invertible key matrix of size $n \times n$
  - **invertible matrix:** an invertible matrix is simply a matrix that has an inverse
  - **n:** the block size
- Encryption
  - the plaintext is divided into blocks of n uppercase ASCII letters
  - each block is represented as a vector of length n
  - the generated key matrix is multiplied by each vector modulo 26, to obtain the corresponding ciphertext vector
    - we use modulo 26 as ASCII upper-case letters are only 26 letters
    - multiplying the key matrix $n \times n$ with the vector $n \times 1$ will result in a vector of size $n \times 1$ which represents the ciphertext vector
  - the ciphertext vectors are converted back into letters to produce the encrypted text
  - if the last block in the plaintext has a length less than n then additional letters are added to the last block as padding to have exact length n, this is mainly to be able to perform matrix multiplication with the key matrix
- Decryption
  - the ciphertext is divided back into blocks of n letters
  - each block is represented as a numerical vector
  - the inverse of the key matrix modulo 26 is computed
    - the inverse matrix modulo 26 is computed using the following formula
      $A^{-1} = (\frac{1}{det(A)} \times adj(A))\ \%\ 26$
    - $det(A)$: determinant of matrix A

- $adj(A)$: Adjugate of matrix A
- the formula can be rewritten as $A^{-1} = (det(A)^{-1} \% 26 \times adj(A)) \% 26$

- the inverse key matrix is multiplied by each vector, modulo 26, to obtain the corresponding plaintext vector
  - we are always guaranteed to get back the plaintext vector, because if you recall from the previous Math Background Section the multiplication of a matrix and its inverse always yields the identity matrix
  - and during encryption, the ciphertext block is equal to $K.A$, and when decrypting we multiply the ciphertext block with the inverse of the key matrix which is $K^{-1}.K.A$, and recall that $K^{-1}.K = I$, so we can say that during decryption we simply multiply the identity matrix with A $I.A$, and recall that $I.A = A$, that is why we always get back our exact plaintext after decryption
    - A: plaintext block
    - K: Key Matrix
    - K^{-1}: Inverse Key Matrix
    - I: Identity Matrix
- the plaintext vector is converted back into letters to produce the decrypted text

One nice fact about Hill cipher is that during encryption if a block has the same letters it doesn't imply that the ciphertext block will have the same exact mappings because during matrix multiplication the same letters can be mapped to different values, which makes it a bit harder to figure out the mapping by just guessing

c. Code Implementation:

We will first describe the 2 classes that are responsible for matrix operations, then we will go through the Hill cipher code logic

Matrix Class:

- The Matrix class mainly represents a generic matrix and provides some methods for basic matrix operations
- **Methods Provided:**
  - __init__(matrixList): constructor that takes in a 2D array to represent a matrix and checks if this 2D array is a valid matrix
  - apply_mod(mod): applies modulo operation to all elements of the matrix
  - apply_constant_multiplication(const): multiplies all elements of the matrix by a constant value
  - transpose(): returns a new Matrix instance that represents the transpose of the current matrix
    - formally the transpose is just flipping the rows and columns
    - $given\ matrix\ A,\ A^{T}_{ij} = A_{ji}\ for\ all\ the\ (i,\ j)\ positions\ in\ A$
  - matrix_multiplication(otherMatrix): returns a new Matrix instance that represents the matrix multiplication of the otherMatrix with the current matrix
    - the code works exactly using the formal definition of matrix multiplication which is
    - $given\ matrix\ A\ with\ dimensions\ N \times M,\ and\ matrix\ B\ with\ dimensions\ M \times K$
    - $C = A.B\ where\ C_{ij} = \sum_{k=1}^{m} (a_{ik}.b_{kj}),\ and\ C\ has\ dimensions\ N \times K$
  - random_matrix(rows, cols, start, stop): is a static method that generates a random matrix of size $rows \times cols$, and the random values are in the range (start, stop)

SquareMatrix Class:

- The SquareMatrix class extends the Matrix class and it is mainly used to represent only Square Matrices and provides some methods for matrix operations specific only to square matrices
- **Methods Provided:**
  - __init__(matrixList): constructor that takes in a 2D array to represent a square matrix and checks if this 2D array is a valid square matrix
  - row_col_elimination(row, col): returns a new SquareMatrix instance representing the sub-matrix of the current matrix after removing the (row, col) from it
  - determinant(): computes the determinant of the square matrix using recursion
    - $we\ can\ define\ a\ recurrence\ for\ computing\ det(A)\ for\ a\ square\ matrix\ A\ of\ size\ N \times N$
    - $det(A) = A_{00}\ iff\ A\ is\ a\ 1 \times 1\ matrix$
    - $det(A) = \sum_{j=0}^{N} (-1)^{j} * A_{0j} * det(A.rowColElimination(0,\ j))$
  - cofactor(): returns a new SquareMatrix instance representing the cofactor of the current matrix
    - $we\ can\ define\ the\ cofactor\ of\ a\ square\ matrix\ A\ of\ size\ N \times N\ as\ the\ following$
    - $cofactor(A)_{ij} = (-1)^{i+j} * det(A.rowColElimination(i,\ j))$

Hill Cipher Implementation:

- The Hill cipher code implementation is exactly as explained in the previous section that described the Hill cipher algorithm steps

- How was the key matrix generated
  - We generated a random matrix of size N, and if it satisfies the following conditions then it is used else we generate another random matrix until the conditions are met
    - condition 1: The key matrix has to be invertible, simply it should have an inverse, and this was handled by the condition $det(A) != 0$
    - condition 2: the $det(A)^{-1}$ has a modular multiplicative inverse and the condition $gcd(det(A), 26) == 1$, as there is proof stating that a modular multiplicative inverse exists if and only if the greatest common divisor between the number **n** and the modulo **m** is 1, and in our case 26 is the modulo and det(A) is the number that we are interested in finding the modular multiplicative inverse for
- Now one thing that wasn't described in the previous section was how we handled non-uppercase letters
  - for non-uppercase letters, we simply didn't encrypt them and just placed all non-uppercase letters in plaintext in the ciphertext in their exact positions as they were in the plaintext
- Final note as described in the Hill cipher algorithm if the last block has a size less than the size of the key matrix we add padding to the last block in the code the letter 'Z' was added as padding if the last block size was small

d. Running the Code:

Here we will go through how to run the code and the expected input/output. The following is the entry point of the code, and the expected input/output:

```
if __name__ == "__main__":
    key_matrix = generate_key_matrix(3, 0, 25)

    plain_text = input("Please enter plain text: ")
    cipher_text = encrypt(plain_text, key_matrix)
    print("Cipher Text: ", cipher_text)

    decrypted_text = decrypt(cipher_text, key_matrix)
    padding = len(decrypted_text) - len(plain_text)
    print("Plain with padding: ", decrypted_text)
    print("Plain: ", decrypted_text[:-padding])
```

```
Please enter plain text: TEST
Cipher Text:  GHHZXB
Plain with padding:  TESTZZ
Plain:  TEST
```

- first to run the code you can use the following command **python "task 2.py"**
- upon running the script the following should happen:
  - the user will be asked to input the plain text
  - after the plain text has been read from the user it will be encrypted and the cipher text will be displayed on the screen
  - then the ciphertext will be decrypted and the decrypted plaintext will be displayed on the screen once with the padding if padding was added, and once after removing the padding

2. Diffie Hellman Man-in-the-Middle Attack:

a. Prerequisite Math Background:

So before going into the task goals and the deffile-hellman algorithm let's just go through some math concepts that will be useful when explaining the diffie-hellman algorithm

Math Background:

So before going into the diffie-hellman algorithm let's give some prerequisite math background

- **Fundamentals of Power:**
  - Product law → $x^a \cdot x^b = x^{ab}$
  - So we can use the product law in the opposite direction

$2 \mid a$ ⇒ This means a is divisble by 2

$$x^a = x^{a/2} \cdot x^{a/2}$$

$2 \nmid a$ ⇒ This means a is not divisble by 2, i.e. 'a' is odd

~~x^a~~ .

$$a = 1 + (a-1) \quad 2 \mid a-1$$

$$x^a = x^1 \cdot x^{a-1}$$

$$= x \cdot x^{(a-1)/2} \cdot x^{(a-1)/2}$$

⇒ we can simply this by writing $x^a = x \cdot x^{\frac{a}{2}} \cdot x^{\frac{a}{2}}$

- Fundamentals of Modular Arithmetic:

  - First you know

$$a \% n = 0 \ldots n-1$$

  - Mod operator can be distributed smoothly among expressions

Addition: $(a+b) \% n = (a \% n + b \% n) \% n$

Multiplication: $(ab) \% n = (a \% n \cdot b \% n) \% n$

Proof: $(a+b) \% n = (a \% n + b \% n) \% n$

writing 'a' in terms of n

$$a = qn + r \qquad q: \text{quotient}, \quad r: \text{remainder}$$

e.g $a = 10 \quad n = 3$

$10 = 3 \times 3 + 1 \quad q = 3, \quad r = 1$

Let's continue:

$$a = q_1 n + r_1 \qquad b = q_2 n + r_2$$

⇒ let's continue proving ~~(a+b)%n~~

$$(a+b)\%n = (a\%n + b\%n)\%n$$

$$a = q_1 n + r_1 \quad b = q_2 n + r_2$$

$$(a+b)\%n = (q_1 n + r_1 + q_2 n + r_2)\%n$$

$$= [(q_1+q_2)n + (r_1+r_2)]\%n$$

Now we know that Mod removes all the cycles from a number, In other words

$$Kn\%n = 0 \quad \text{for } \forall K \in \mathbb{Z}$$

So $(q_1+q_2)n\%n = 0$

So we can say

$$[(q+q_2)n + r_1+r_2]\%n =$$

$$(r_1+r_2)\%n$$

recall that Mod computes the remainder, So

$$r1 = a\%n \quad r2 = b\%n$$

and that concludes our proof that

$$(a\%n + b\%n)\%n = (a+b)\%n$$

⇒ You can use the same principles to proof that for the case of Multiplication

$$(a \cdot b) \% n = (a \% n \cdot b \% n) \% n$$

- So we have talked a lot about Modular Arithmetic, now let's Combine Mod with Powers

$a^b \% n$ ~~we know that~~

we know that $a^b = a^{b/2} \cdot a^{b/2}$ iff $2 \mid b$

, or $a^b = a \cdot a^{\lfloor b/2 \rfloor} \cdot a^{\lfloor b/2 \rfloor}$ iff $2 \nmid b$

So $a^b \% n = (a^{b/2} \cdot a^{b/2}) \% n$ iff $2 \mid b$

$= (a^{b/2} \% n \cdot a^{b/2} \% n) \% n$

That is a very very Imp Concept to understand about Mod with Powers

~~Fact 1: if we know the value~~ $a^{b/2} \% n$

~~Proof~~

⇒ Some facts about Mod and Power.

we have just shown that

$$a^b \% n = (a^{b/2} \% n \cdot a^{b/2} \% n) \% n$$

iff $2 | b$

AND

$$a^b \% n = (a \% n \cdot a^{\lfloor \frac{b}{2} \rfloor} \% n \cdot a^{\lfloor \frac{b}{2} \rfloor} \% n) \% n$$

iff $2 \nmid b$

Fact 1: We only need to Compute $a^{b/2} \% n$ to be able to know the value of $a^b \% n$

Fact 2: we can divide $a^{b/2} \% n$ further to $a^{b/4} \cdot a^{b/4}$

given the above 2 facts this implies that we can calculate

$a^b \% n$ in only $\log_2(b)$ steps

⇒ So $a^b \% n$ can be computed very efficiently even for very large values for the exponent

Bonus :

- Here is a pSeudo Code to show how $a^b \% n$ can be Computed
- The following Code is recursive

```
pow_mod(a, b, n) {
    if (b == 1) // base Case
        return a%n;

    SubResult = pow_mod(a, b/2, n);
    SubResult = subResult * subResult;

    if (b%2 != 0) //odd exponent
        SubResult = SubResult *
                    (a%n)

    return SubResult%n;
}
```

b. <u>Task Goal & Algorithm:</u>

- The task goal was to illustrate how 2 users on the internet can communicate together using the Diffie-Hellman key exchange algorithm, and also show how a man-in-the-middle attack can work using the Diffie-Hellman key exchange protocol
- Diffie-Hellman uses a symmetric key between the 2 users to encrypt and decrypt messages, by symmetric keys I mean that they are identical, but what makes Diffie-Hellman special is that it

generates the symmetric keys between the 2 users using asymmetric private-public key pairs. The motivation behind that is that the 2 users communicate over an insecure public channel which means the symmetric key should never go through the communication channel between the 2 users

- Now that feels a bit abstract let's briefly describe how the deffie-hellman algorithm works in detail:
  - the following notations will be used
    i. private key: k
    ii. public key: m
    iii. generator: g
    iv. prime modulus: p
    v. secret key: s

  - 'g' and 'p' are agreed between the 2 users

  - user 1 does the following:
    - generates a random private key k_1
    - computes his public key using the following formula
      - $m_1 = g^{k1} \% p$
      - recall from the math background section that it takes $log_2(k1)$ to compute modular exponentiation
    - user 1 sends m_1 to user 2
    - user 1 receives m_2 from user 2
    - user 1 generates the secret key using the following formula
      - $s_1 = (m_2)^{k_1} \% p$
      - recall $m_2 = g^{k2} \% p$
      - $s_1 = (g^{k2} \% p)^{k_1} \% p$

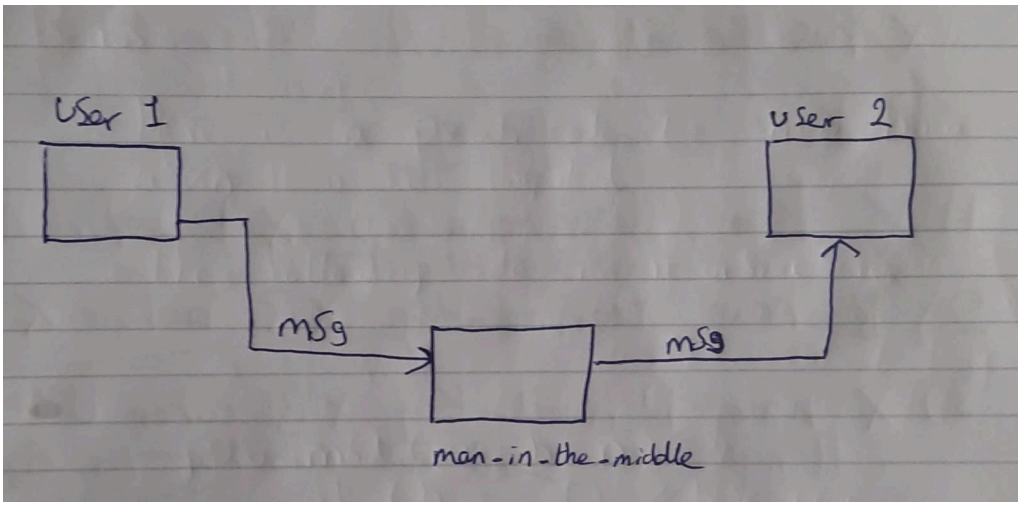
  - user 2 does the following:
    - generates a random private key k_2
    - computes his public key using the following formula
      - $m_2 = g^{k2} \% p$
    - user 2 sends m_2 to user 1
    - user 2 receives m_1 from user 1
    - user 2 generates the secret key using the following formula
      - $s_2 = (m_1)^{k_2} \% p$
      - recall $m_1 = g^{k1} \% p$
      - $s_2 = (g^{k1} \% p)^{k_2} \% p$

  - now the magic is that both secret keys computed at the 2 users are exactly equal, now let's try to proof why the secret keys are equal
    - first let's proof the following statement $(a^b \% n)^c \% n = a^{bc} \% n$
      - let's start from the right-hand-side of the equation $a^{bc} \% n$
      - $a^{bc} = (a^b)^c$
      - $(a^b)^c = (a^b * a^b * a^b * .....)$ we keep multiplying a^b to itself 'c' amount of times
      - recall that mod can be distributed among values that or added or multiplied recall that from the math background section
      - $a^{bc} \% n = (a^b)^c \% n = (a^b * a^b * a^b * .....) \% n$
      - now let's apply $\% n$ for every term in our above expression
      - $(a^b * a^b * a^b * .....) \% n = (a^b \% n * a^b \% n * a^b \% n * .....) \% n$
      - this is equivalent to $(a^b \% n)^c \% n$
      - so that concludes our proof that $(a^b \% n)^c \% n = a^{bc} \% n$

    - now that we have proven the above statement it becomes easy to show that the secretes keys are equal
      - recall $s_1 = (m_2)^{k_1} \% p = (g^{k2} \% p)^{k_1} \% p$
      - recall $s_2 = (m_1)^{k_2} \% p = (g^{k1} \% p)^{k_2} \% p$
      - so based on the above statement, we can rewrite both secret keys as follows

- $s_1 = (g^{k_2} \% p)^{k_1} \% p = g^{k_2 * k_1} \% p$
- $s_2 = (g^{k_1} \% p)^{k_2} \% p = g^{k_1 * k_2} \% p$
- $g^{k_2 * k_1} = g^{k_1 * k_2}$, and that concludes our proof that $s_1 = s_2$

- Another goal of the task is to show how a man-in-the-middle attack can be used on the Diffie-Hellman key exchange protocol and is quite simple instead of 2 users exchanging their public keys together they will exchange their public keys with a man-in-the-middle and whenever a user sends a message to another user the message first will be intercepted by the man-in-the-middle and decrypted using the sender secret key then decrypted using the receiver secret key and sent to the receiver from the man-in-the-middle acting as the normal sender. That might feel a bit abstract so the below image describes how a man-in-the-middle attack works



- Let's briefly describe the steps that take place for a man-in-the-middle attack with diffie-hellman:
  - first user 1 attempts to send his public key to user 2
  - the man-in-the-middle intercepts the message. Instead, he sends his public key to user 2 and acts to be user 1 when sending his public key to user 2. Also, the man-in-the-middle computes the secret key using user 1 public key
  - once user 2 receives the public key of the man-in-the-middle he computes the secret key using the public key of the man-in-the-middle
  - user 2 then sends his public key to user 1
  - the man-in-the-middle intercepts the message. Instead, he sends his public key to user 1 and acts to be user 2 when sending his public key to user 1. Also, the man-in-the-middle computes another secret key using the user 2 public key
  - once user 1 receives the public key of the man-in-the-middle he computes the secret key using the public key of the man-in-the-middle
  - after this process, user 1 and the man-in-the-middle share the same secret key denoted by s1
  - and user 2 and the man-in-the-middle share the same secret key donated as s2
  - Now when user 1 wants to send a message to user 2 he will not be able to, because both users use different keys
  - user 1 first encrypts the message he wants to send using s1
  - the man-in-the-middle intercepts the message sent from user 1 and decrypts it using s1
  - the man-in-the-middle then encrypts the message using s2 and sends it to user 2
  - user 2 receives the message and decrypts it using s2
  - One important note is that the man-in-the-middle can change the content of the message before encrypting it using s2 and sending it to user 2

c. <u>Code Implementation:</u>

In this section, we will go through a high-level overview of the code implementation for the Diffie-Hellman key exchange protocol and the man-in-the-middle attack. The implementation mainly involves 2 main classes "InternetUser" and "Interceptor"

- **InternetUser Class:**
  - **Purpose:** The InternetUser class represents any user that wants to communicate over the internet
  - **Key Generation:** internet users initially generate their private and public keys using the following parameters: dh.generate_parameters(generator=2, key_size=2048)

- **Key Exchange:** public keys are exchanged between internet users, and a shared secret key is derived using the Diffie-Hellman algorithm
- **Message Communication:** internet users can send encrypted messages to each other using the derived shared secret key
- **Interceptor:** an instance of the InternetUser class may contain an instance of an interceptor, and when an interceptor is set key exchange, and message communication are always redirected to the interceptor and not the receiver

- **Interceptor Class:**
  - **Purpose:** Extending the functionalities provided by the InternetUser Class to allow for a man-in-the-middle attack
  - **Key Exchange Interception:** The interceptor intercepts the key exchange process, by sending his public key to the 2 users and sharing a secret key with each user
  - **Message Interception:** Intercepted messages are decrypted using the sender key and re-encrypted using the receiver key. Optionally we can make the interceptor modify messages transmitted between the 2 users

- For encryption, we use the simple Caesar cipher algorithm which just replaces every character in the plain text with another character based on the encryption key, the encryption and decryption process can be defined as follows:
  - encryption(plaintext, key): $ciphertext = (c + key) \% 128 \ for\ each\ character\ in\ the\ plaintext$
  - decryption(ciphertext, key):
    $plaintext = (c - key) \% 128 \ for\ each\ character\ in\ the\ ciphertext$

d. Running the Code:

Here we will go through how to run the code and the expected output. The following is the entry point of the code, and the expected output:

```python
if __name__ == "__main__":
  alice = InternetUser("alice_ip")
  bob = InternetUser("bob_ip")

  darth = Interceptor("darth_ip", evil=True) # set evil to False to only eavesdrop messages sent
  darth.addTargetedInternetUser(alice) # remove this line to allow secure communication between alice & bob
  darth.addTargetedInternetUser(bob) # remove this line to allow secure communication between alice & bob

  alice.sendKey(bob)
  bob.sendKey(alice)

  print("Equal derived keys: ", alice.derived_key == bob.derived_key)
  print(darth.derived_keys[alice.ip] == alice.derived_key) # remove this line if an interceptor is not set
  print(darth.derived_keys[bob.ip] == bob.derived_key) # remove this line if an interceptor is not set

  alice.sendMessage(bob, "Hello Bob, how are you doing")
  print()
  bob.sendMessage(alice, "Alice have you hacked me!!!")
```

```
Received Public Key from:  alice_ip
Received Public Key from:  bob_ip
Equal derived keys:  False
True
True
Intercepted Message:  Hello Bob, how are you doing
Message Received:  Your device have been hacked :(
        - Received From:  alice_ip

Intercepted Message:  Alice have you hacked me!!!
Message Received:  Your device have been hacked :(
        - Received From:  bob_ip
```

- first to run the code you can use the following command **python "task 2.py"**
- upon running the script the following should happen:
  - 2 internet users, Alice and Bob, are created
  - An interceptor called Darth is created
    - Darth is configured to intercept communication between Alice and Bob
    - this simply means that Alice and Bob will have their interceptor instance set to Darth, and that means that any communication between Alice and Bob will be intercepted by Darth
  - Alice sends her public key to Bob, but because Alice has Darth set as an interceptor Alice's public key will be automatically sent to Darth instead

- Similarly, Bob's public key will be intercepted by Darth when sending his public key to Alice
- The derived keys between Alice and Bob should not match since there was no direct key exchange
- The derived keys between Alice and Darth should match, and the derived keys between Bob and Darth should also match
- Alice sends a message to Bob saying "Hello Bob, how are you doing", however Darth intercepts and modifies Alice's message before forwarding it to Bob.
    - Darth, being configured as an evil interceptor, replaces Alice's original message with "Your device has been hacked :(" and sends it to Bob, using Alice's IP address to hide his real identity
- Once Bob receives the message from Darth he sends a message to Alice saying "Alice have you hacked me!!!", however Darth also intercepts Bob's message before forwarding it to Alice
    - Darth, being configured as an evil interceptor, replaces Bob's message with "Your device has been hacked :(" and sends it to Alice, using Bob's IP address to hide his real identity

3. Elliptic Curve Digital Signature Algorithm:

a. Task Goal & Algorithm:

The task goal was to create a digital signature using Elliptic Curve Cryptography (ECC). In this section we will define how digital signatures are created, then we will later briefly describe how can private and public keys be generated using elliptic curves.

Digital Signatures:

- First, what is a digital signature? A digital signature is a technique to verify the authenticity of the source of a message from a sender
- Now how are digital signatures created, and how do they work
    - it all starts with a sender who would like to send some data, but he needs to sign this data somehow to verify that he is the source of the message
    - so first thing is that he hashes the data he wants to sign using a cryptographic hash function, a cryptographic hash function has the following properties
        - have very low probability that 2 pieces of data hash to the same value
        - one-way function which means that given the hash value, you cannot get the data that produced this hash value back
    - then the hashed data is encrypted using the sender's private key and the result of encrypting the hash value is called the digital signature
    - and the sender sends his digital signature to the public
    - now if the receiver needs to verify if the sender's identity is authentic all they need to do is use the sender's public key to decrypt the digital signature to get the hash value, and then hash the sender data and compare the two hash values if they are equal that means that the sender identity is authentic
    - Here we have done 2 checks at the same time first is that the plain data at the receiver side is identical to the sender data that he was intending to send and it wasn't played with during the transmission, and the public key used for decryption is the one belonging to the sender who owns the corresponding private key and no one else

Elliptic Curve Cryptography:

- Recall that we said that private-public keys are used in the process of digital signatures
- Now elliptic-curve cryptography is an asymmetric key generation algorithm that produces private-public key pairs
- without delving into a lot of math elliptic curve initially generates a very large random private key number
- this random private key random is used as the number of additions that will be done for a point on an elliptic curve equation. The addition operation used in elliptic curve cryptography is not the usual addition that we are used to, here is how it works
    - first, we start with a fixed (x, y) point on the elliptic curve let's call it **g**
    - we add the start point to itself to get another point $g + g = 2g$
    - then we add the start point to the previous result so we get $g + 2g = 3g$
    - the addition is performed by drawing a line between the 2 points that are being added and reflecting the intersection point on the curve, and that reflected point of intersection on the curve is the result of the addition
- so after generating a very large private key let's call it **p** the public key is equal to adding **g** to itself **p** times $public\ key = pg,\ which\ is\ an\ (x, y)\ coordinate\ on\ the\ elliptic\ curve$

b. Code Implementation:

We will mainly go through the 4 functions in the code and their purpose:

- generate_keys(): generates private and public keys using elliptic curves cryptography
- sha256_hash(text): hashes plain text using sha256 hash function

- sign(priv_key, hash_digest): return a digital signature using the private key of the sender and the hash value of the message
- verify_signature(signature, text, pub_key): return a boolean value indicating if the digital signature is authenticated or not using the sender signature, public key, and plain text

c. Running the Code:

Here we will go through how to run the code and the expected output. The following is the entry point of the code, and the expected output:

```
if __name__ == "__main__":
  # Generating Keys
  priv_key, pub_key = generate_keys()
  print("private key: ", priv_key.private_numbers().private_value)
  print("public key (x, y): ", pub_key.public_numbers().x, pub_key.public_numbers().y)
  print()

  # Hashing Plain Text
  plain_text = "LZSCC.363"
  hash_value = sha256_hash(plain_text)
  print("SHA-256 hash of '{}': {}".format(plain_text, hash_value.hex()))
  print()

  # Creating the digital signature
  signature = sign(priv_key, hash_value)
  print(signature)
  print()

  # Verifying the digital signature
  verified = verify_signature(signature, plain_text, pub_key)
  print("Verified: ", verified)
```

```
private key:  65655269424471630817089218036175478083150494260419060729735473551250483228291731797364990137680548324080666050141 31
public key (x, y):  11042365331402773760534775555083219191163940819912342268882950661084616029659233641947431133550332246741016067 093237 85611392795205782397922545002727159730174274442881488082295237480650188107782706093899551955058394973882959533492 38

SHA-256 hash of 'LZSCC.363': e5075a092dd9029ce87242e31f86ad17a5c3bc3519a7406f556efdde7ca17067

b'0e\x020=d\xf3\x90\xd3\x96G\xceq \xeb\xb8AR2\xc2Y\xc8\xd8?\xe6\n\x9f4\xcf\x8d\x8b\x8d\xf2)S\x90\xe9\x9b\x141S\x8c\xac\xb3\xc1\x02\x7f\x17\xed>\xac\xed\x021\x00\xdb\x99\\%T\xa0%\xd9\x11S\xaa\xdc\x8a\xa9ma\x1b\xdd\\{K\x92\x058`\x9f\x07\xaf\x06\xea\xfe<\xf4\xba~\x8e\xb3h\x0f{\xe9\xae\x19F\x13P\x9a\xde'

Verified:  True
```

- first to run the code you can use the following command **python “task 3.py”**
- upon running the script the following should happen:
  - the private key of the sender should be printed
  - the public key (x, y) coordinate returned from the elliptic curve algorithm should be printed
  - the hash value of the plain text should be printed
  - the digital signature of the sender should be printed in binary format
  - True/False indicating if the digital signature is verified or not